Bujar Sefa
CSC 21200 -BC
Professor Liu
HW #2

Overview of Homework:
This homework had to do with writing various functions related to character arrays - 1D arrays, 2D arrays, static arrays, and dynamic arrays. Functions varied from storing data (input functions) to manipulating data(shifting/rotating), to generating more data (such as an enlarged 2D array.)

1. takeUserInputRetSize():
What: This function stores user input into a static 1D char array.
How: This function works by using the cin function to input user alphabetic characters and storing it into a static array, that is passed by reference. The number of characters is also stored so that it can be utilized in later functions. The function uses the isalpha command so that the array is guaranteed to be alphabetical and cannot be greater than the MAXSIZE, which is 100.
Pre-Condition: User input must be alphabetical and cannot be greater than the MAXSIZE (100).
Post-Condition: A static char array with the user input is stored and so is its size.
Worst Time Complexity: O(n)

```
void takeUserInputRetSize(char arr[], size_t& size){
    char c ;
    size = 0;
    while(size < MAXSIZE && cin >> c && isalpha(c)){
        arr[size] = c;
        size++;
    }
}
```

2. highestOccruances():
What: This function finds the character that is found the most in the array. If two characters have the same highest occurance, the one that is alphabetically closes to a is returned.
How: This function works by going through the whole array and counting the number of occurrences per character, and removing those characters from the array. If that letter is closer to a, then it is stored and returned.
Pre-Condition: (Assumed all lowercase letters).
Post-Condition: The highest number of occurrences of the letter closes to a is returned.
Worst Time Complexity: O(n^2)

```
size_t highestOccurances(char arr[ ], size_t size){
    assert(size <= MAXSIZE && size > 0);
    size_t hOccurrance = 0;
    char letter = '{';
    size_t countingOccurrance = 0;
    for(size_t i = 0; i < size; i++){
        countingOccurrance= 0;
        countingOccurrance++;
        for(size_t j = i+1; j < size; j++){
               if(arr[i] == arr[j]){
                      countingOccurrance++;
                      arr[j] = arr[size-1];
                      size--;
                      j--;
               }

        }
        if(countingOccurrance >= hOccurrance){
               if((int)letter > (int)arr[i]){
                      letter = arr[i];
                      hOccurrance = countingOccurrance;
               }
        }
    }
    return hOccurrance;
}
```

3. shiftArray():

What: This function shifts an array based on the number of shifts the user inputs.

How: This function works by taking in a shift value from the user, checking if the shift value is in reverse or greater than the size and molding it so that it can be a regular shift (ie if its negative, Size + shift, if it's greater than the size, mod). It then swaps all the characters in the array. The it swaps the characters from the start to the shift scale and all the characters from the shift scale to the end. Finally the shifted array is ‘returned’ (just stored by reference).

Pre-Condition: Shift must be an integer value.

Post-Condition: The shifted array is stored based on the number of shifts a user inputs.

Worst Time Complexity: O(n)

```
void shiftArray(char arr[], size_t size, int shift){
    assert(size<=MAXSIZE && size >0 );
    cin >> shift;
    if(abs(shift)>size){
        shift = shift%size;
    }
    if(abs(shift)==(-shift)){
        shift = size + shift;
    }
    for(size_t i = 0; i < size/2; i++){
        char temp = arr[i];
        arr[i] = arr[size-1-i];
        arr[size-1-i] = temp;
    }
    for(size_t i = 0; i <= (shift-1)/2; i++){
        char temp = arr[i];
        arr[i] = arr[shift-1-i];
        arr[shift-1-i] = temp;
    }
    size_t z =1;
    for(size_t i = shift; i < shift+((size-shift)/2); i++){
        char temp = arr[i];
        arr[i] = arr[size-z];
        arr[size-z] = temp;
        z++;
    }
 }
```

4. appendAndSort():

What: This function returns a new and sorted array that is the made by appending two other arrays.

How: This function works by comparing the values in two static char arrays at a time and storing them into a new static array. If compares characters by looking at its ASCII values. Once one array is fully stored in the char array, the rest of the values are then put in from the left over array (thats if their sizes are unequal). Also note, that the arrays combined cannot be greater than the MAXSIZE or then the function wouldn't compute.

Pre-Condition: (Assumed char arrays are sorted). The size of the two arrays combined cannot be greater than the MAXSIZE.

Post-Condition: A sorted array of the two appeneding arrays given by user is returned.

Worst Time Complexity: O(n)

```
char* appendAndSort(const char* arr1, const size_t arr1Size, const char* arr2, const size_t
arr2Size, int& arrSize){
    assert((arr1Size + arr2Size) <= MAXSIZE);
    char* arr = new char[arr1Size+arr2Size];
    size_t arrI = 0, arr1I = 0, arr2I = 0;
    while(arr1I < arr1Size && arr2I < arr2Size){
        if((int)arr1[arr1I] < (int)arr2[arr2I] ){
               arr[arrI++] = arr1[arr1I++];
        }
        else{
               arr[arrI++] = arr2[arr2I++];
        }
    }
    while(arr1I < arr1Size){
        arr[arrI++] = arr1[arr1I++];
    }
    while(arr2I < arr2Size){
        arr[arrI++] = arr2[arr2I++];
    }
    arrSize = arrI;
    return arr;
}
```

5. mNMatrix():

What: This function asks users for a input for row and column sizes for a 2D matrix/array and users are then asked to fill it up.

How: This function works by taking in user input for the row and column and asserting that it is not greater than the MAXSIZE. Then the user can enter characters that will be entered into the array in order of input into the 2D matrix/array. The dimensions and the 2D matrix are then returned/saved.

Pre-Condition: The size of the row and column must be less than or equal to the MAXSIZE. (So can be a total of 100*100 matrix/2d Array)

Post-Condition: A matrix/2D array defined in size by the user, is returned with user input.

Worst Time Complexity: O(n^2)

```
char** mNMatrix(size_t& x, size_t& y){
    size_t m;
    size_t n;
    cin >> m;
    cin >> n;
    char c;
    assert(m <= MAXSIZE && m > 0 && n <= MAXSIZE && n>0 );
    char **matrix;
    matrix = new char*[m];
    for(int i = 0; i < m; i++){
        matrix[i] = new char[n];
    }
    int row = 0;
    int column = 0;

    while(row<m){
        column = 0;
        while(column<n && cin>>c){
               matrix[row][column] = c;
               column++;
        }
        row++;
    }
    x = m;
    y = n;
    return matrix;
}
```

6. mNMatrix():

What: This function rotates a matrix by intervals of 90 degrees, if interval is + it is rotated counterclockwise, if - it is rotated clockwise.

How: This function works by taking in user input for the number of 90 degree rotations, both counter/clockwise. The interval is then adjusted so that each will fall in the interval of 0-3 (- values are converted by 4-rotation, larger than 4 values are modded). Then each value in the array is rotated working its way from the outside to the inside.

Pre-Condition: The number of rotations must be an integer. (Matrix that is unputed is assumed to be a square -- though this is checked).

Post-Condition: A rotated matrix in intervals of 90 degrees is 'returned' (saved back as function is void).

Worst Time Complexity: O(n^2)

```
void rotateMatrix(int rotation, char** arr, const size_t& m, const size_t& n){
        if(m!=n) return;
        cin >> rotation;
        if(abs(rotation)>4){
                rotation = rotation%4;
        }
        if(abs(rotation)==(-rotation)){
                rotation =  4 + rotation;
        }
        for(int r = 0; r < rotation; r++){
                for(int i = 0; i < m/2; i++){
                        char temp = arr[i][i];
                        arr[i][i] = arr[i][m-1-i];
                        arr[i][m-1-i] = arr[m-1-i][m-1-i];
                        arr[m-1-i][m-1-i] = arr[m-1-i][i];
                        arr[m-1-i][i] = temp;
                }
        }
}
```

7. enlargeMatrix():

What: This function takes a m*n matrix and enlarges it by a scale size that is given by the user.

How: This function works creating a dynamically allocated array that is the size of the imputed dimensions * the scale. Then it goes through the inputted array and inputs each element/character into the enlarged array by the size of scale both column and row size. It repeats for all characters.

Pre-Condition: The size of the scale inputted by user must be greater than zero and cannot be greater than MAXSIZE. It must be an integer value.(size_t)

Post-Condition: A enlarged matrix dictated by user input is returned.

Worst Time Complexity: O(n^3)

```
char**  enlargeMatrix(size_t scale, char** originalMatrix, size_t originalRow, size_t
originalColumn, size_t& enlargedRow, size_t& enlargedColumn){
    cin >> scale;
    assert(scale <= MAXSIZE && scale >0);
    enlargedRow = scale*originalRow;
    enlargedColumn = scale*originalColumn;
    char** enlargedMatrix = new char*[enlargedRow];
    for(size_t i = 0; i <(enlargedRow); i++){
        enlargedMatrix[i] = new char[enlargedColumn];
    }
    /*Indices for enlarged array*/
    size_t iEnlarged = 0, jEnlarged = 0;
    /*Indices for keeping track of how much expansion there is*/
    size_t endR = scale;
    size_t endC = scale;
    /*Indicides for original array*/
    size_t iOriginal = 0, jOriginal = 0;
    for(int i = 0; i < originalRow; i++){
       int j = 0;
       for(int a = 0; a < scale; a++){
            for(int b = 0; b < enlargedColumn; b++){
              if(iEnlarged < enlargedRow && jEnlarged < enlargedColumn){
                     enlargedMatrix[iEnlarged][jEnlarged] = originalMatrix[i][j];
                     jEnlarged++;
                     if((b+1)%scale==0){
                        j++;
                     }
              }
              j-=(originalColumn);
              jEnlarged-=enlargedColumn;
              iEnlarged++;
       }
    }
    return enlargedMatrix;
}
```

Additional Questions/Notes:

1. Using isAlpha for mNMatrix():

Improvements would definitely be for the entering into a m*n matrix (question five) so that all inputs are alphabetical character, but couldn't see to bypass a bug.

Header File: Sefa_Bujar_HW2.h:

```
#include <iostream>
#include <cassert>
#include <cctype>
#include <cmath>
#include <cstdio>
#include <cstdlib>
using namespace std;


static const size_t MAXSIZE = 100;

void takeUserInputRetSize(char arr[], size_t& size);
size_t highestOccurances(char arr[], size_t size);
void shiftArray(char arr[], size_t size, int position);
char* appendAndSort(const char* arr1, const size_t arr1Size, const char* arr2, const size_t
arr2Size, int& arrSize);
void rotateMatrix(int rotation, char** arr, const size_t& m, const size_t& n);
char** enlargeMatrix(size_t size, char** c, size_t m, size_t n, size_t& x, size_t& y);
char** mNMatrix(size_t&x, size_t& y);
```